Maven: A Data-Parallel Architecture for Par Lab

Yunsup Lee
Christopher Batten (Now at Cornell)
Rimas Avizienis
Christopher Celio
Alex Bishara
Richard Xia
Krste Asanovic

Par Lab Winter Retreat 2010
**Goal**: To Build an Energy Efficient and Flexible Data-Parallel Core
Motivation

**Architectural Patterns**
- Multithreading (MT)
- Traditional Vector (TVEC)
- Single-Instruction Multi-Threading (SIMT)
- Vector-Threading (VT)

**Maven Single-Lane VT Core**

**Maven Evaluation**

**Conclusion**
Architectural Design Patterns: Multithreading & Traditional Vector

Programmer’s View (Programming Model)

Multithreading

Memory

Traditional Vector

Machine Implementation

Instruction Memory

Vector Issue Unit

Vector Mem Unit

Data Memory

Instruction Mem
Architectural Design Patterns: Single-Instruction Multi-Threading & Vector-Threading

SIMT (GPU Style)

Vector-Threading

Memory

Instruction Mem

Vector Issue Unit

Lane0

Lane1

Mem Coalescing

Data Memory

CP

Vector Issue Unit

Lane0

Lane1

Vector Mem Unit

Data Memory

Programmer’s View

(Programming Model)

Machine Implementation
Architectural Design Pattern’s Energy vs. Performance Plot
Motivation

Architectural Patterns

**Maven Single-Lane VT Core**
- Maven Instruction Set Architecture
- Maven μArchitecture
- Maven Programming Methodology

Maven Evaluation

Conclusion
Maven Instruction Set Architecture

Goal: Same MIPS-like instruction set for both CP and μTs

CP specific instructions
- Vector configuration commands
- Vector memory commands
- Vector fetch command
- Move scalar to vector register
- Operating system instructions

Standard MIPS features
- CP and μT same calling convention
- Conditional moves
- Full single-precision floating point

Changes to basic MIPS ISA
- Unify integer and floating point ops
- Eliminate special mul/div registers
- Atomic memory operations
Maven μArchitecture: Single-Lane Vector Units

- One Control Processor with 10-100s of Lanes
- Several Control Processors Each with a Couple Lanes
- Many Control Processors Each with a Single Lane
Maven μArchitecture: Baseline Traditional Vector

loop:
  vload A, a_ptr
  vload B, b_ptr
  vadd C, A, B
  vstore C, c_ptr
  a_ptr++
  b_ptr++
  c_ptr++
  branch
Three Key Microarchitectural Mechanisms

- Loop:
  - \( \text{vload } A, \text{ a_ptr} \)
  - \( \text{vfetch } \text{ut_code} \)
  - \( \text{a_ptr}++ \)
  - \( \text{branch} \)
  - \( \text{ut_code} : \text{branch } a = 0 \)
  - \( \text{op0} \)
  - \( \text{op1} \)
  - \( ... \)
  - \( \text{stop} \)

- Control Processor Embedding
- Vector Fetched Branch Divergence Management
- Density-Time Execution

All three mechanisms are simpler to implement in single-lane vector units

Density-Time Tradeoffs
- More complicated but higher performance
- Completely avoids any work for inactive \( \mu T \)s
Maven Programming Methodology: Compiler Support

Goal: Minimum changes to standard scalar compiler to enable a high-level explicitly data-parallel programming methodology

1. Start with most recent GCC toolchain (4.4.1) with MIPS32 backend
2. Change MIPS32 backend to support unified integer and floating-point registers, add new multiply and divide instructions, and remove unsupported instructions
3. Modify SIMD extensions to support much longer vector
4. Add support for vector registers with standard register allocator
5. Add intrinsics for vector commands
6. Add Maven pipeline model and ability to tune any function for either the control processor or a micro-thread
Maven Programming Methodology: Library Support

```c++
void vvadd_vt( int dest[], int src0[],
               int src1[], int size )
{
    int vlen = vt::set_vlen( size );
    for ( int i = 0; i < size; i += vlen )
    {
        vlen = vt::set_vlen( size - i );

        vt::HardwareVector<int> vsrc0, vsrc1;
        vsrc0.load( &src0[i] );
        vsrc1.load( &src1[i] );

        vt::HardwareVector<int> vdest;

        VT_VFETCH( (vdest), (vsrc0,vsrc1),
                   ({
                      vdest = vsrc0 + vsrc1;
                   }));

        vdest.store( &dest[i] );
    }
}
```

Compiles to instruction which abstracts hardware vector length

C++ class that uses compiler vector types to enable automatic vector register allocation

Unit-stride vector loads

Compiles to separate function which is then vector fetched

Inside vector fetched block vector registers appear as scalar values

Unit-stride vector store
Maven Programming Methodology: Library Support

```c
void vvadd_vt( int dest[], int src0[],
               int src1[], int size )
{
    int vlen = vt::set_vlen( size );
    for ( int i = 0; i < size; i += vlen )
    {
        vlen = vt::set_vlen( size - i );

        vt::HardwareVector<int> vsr0, vsr1;
        vsr0.load( &src0[i] );
        vsr1.load( &src1[i] );

        vt::HardwareVector<int> vdest;

        VT_VFETCH( (vdest), (vsr0,vsr1),
                   {
                       vdest = vsr0 + vsr1;
                       if ( vdest < 0 )
                           error(vdest);
                   ...
                   });
}
```

VT library supported on maven and emulated on native platform

Enables rapid application development using native emulation

Standard C++ allowed inside vector-fetched block including objects, conditionals, loops, and function calls

Currently system calls and exceptions not supported
Motivation

Architectural Patterns

Maven Single-Lane VT Core

**Maven Evaluation**

Conclusion
Evaluation Methodology

Software Toolflow

- C++ Application
  - Native Compiler
    - Native Binary
      - Maven Compiler
        - Maven Binary
          - Maven ISA Sim

Hardware Toolflow

- Verilog RTL
  - Verilog Simulator
  - Synthesis Place&Route
    - Gate-Level Model
      - Layout
        - Verilog Simulator
          - Switching Activity
            - Power Analysis
              - Power

Results

- Area & Cycle Time
- Cycle Count
- Power

TSMC 65nm Process with Synopsys CAD Toolflow
Caveats

- **Half baked results**: Early stage results
  - Instruction fetch energy not considered
  - Data access energy not considered
  - Both penalizes results for the vector machines
- SIMT (GPU Style) machines are approximated with VT machines
- Irregular data parallel microbenchmarks (masked filter, binary search) for the traditional vector machine are hand-coded assembly
  - Other microbenchmarks are all compiled
Evaluated Data-Parallel Core Designs

<table>
<thead>
<tr>
<th>Physical Regs</th>
<th>Multi-threaded</th>
<th>Single-Lane Traditional Vector</th>
<th>Multi-Lane Traditional Vector</th>
<th>Single-Lane Vector-Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>MT0:</td>
<td>μT0:</td>
<td>CP</td>
<td>CP</td>
</tr>
<tr>
<td>Physical Regs</td>
<td>MT1:</td>
<td>μT1:</td>
<td></td>
<td>μT0:</td>
</tr>
<tr>
<td>64</td>
<td>MT0:</td>
<td>μT0:</td>
<td>CP</td>
<td>CP</td>
</tr>
<tr>
<td>Physical Regs</td>
<td>MT1:</td>
<td>μT1:</td>
<td></td>
<td>μT0:</td>
</tr>
<tr>
<td>128</td>
<td>MT0:</td>
<td>μT0:</td>
<td>CP</td>
<td>CP</td>
</tr>
<tr>
<td>Physical Regs</td>
<td>MT1:</td>
<td>μT1:</td>
<td></td>
<td>μT0:</td>
</tr>
<tr>
<td>256</td>
<td>MT0:</td>
<td>μT0:</td>
<td>CP</td>
<td>CP</td>
</tr>
<tr>
<td>Physical Regs</td>
<td>MT1:</td>
<td>μT1:</td>
<td></td>
<td>μT0:</td>
</tr>
</tbody>
</table>
Single-Lane Vector-Thread Unit w/ 256 Registers

- Control Processor: 8.1%
- Vector Register File: 56.9%
- Vector Integer ALUs: 9.7%
- Vector FPUs: 9.4%
- Vector Memory Units: 7.6%
- Other: 8.3%
Microbenchmark: Complex Multiply
Microbenchmark: Masked Filter
Motivation

Architectural Patterns

Maven Single-Lane VT Core

Maven Evaluation

Conclusion
## Conclusion

<table>
<thead>
<tr>
<th>Performance</th>
<th>Energy Efficiency</th>
<th>Application Space</th>
<th>Programming Difficulty</th>
<th>Compiler Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>TVEC, VT</td>
<td>TVEC, VT</td>
<td>VT, SIMT</td>
<td>MT, SIMT</td>
<td>VT, SIMT</td>
</tr>
<tr>
<td>SIMT</td>
<td>SIMT</td>
<td>VT, SIMT</td>
<td>VT</td>
<td>MT</td>
</tr>
<tr>
<td>MT</td>
<td>MT</td>
<td>TVEC</td>
<td>TVEC</td>
<td>TVEC</td>
</tr>
<tr>
<td>Worst</td>
<td>Low</td>
<td>Narrow</td>
<td>Hard</td>
<td>Hard</td>
</tr>
</tbody>
</table>
Future Work

**Short Term**
- Optimize vector control overhead
- Explore banked register file designs
- Evaluate impact of density time execution
- Experiment with application kernels
- Fab a test chip

**Long Term**
- Investigate tightly integrating general-purpose cores with vector-thread data-parallel cores
- SEJITS backend for Maven
Thank you for your attention!
Any Questions?

Yunsup Lee
yunsup@cs.berkeley.edu